



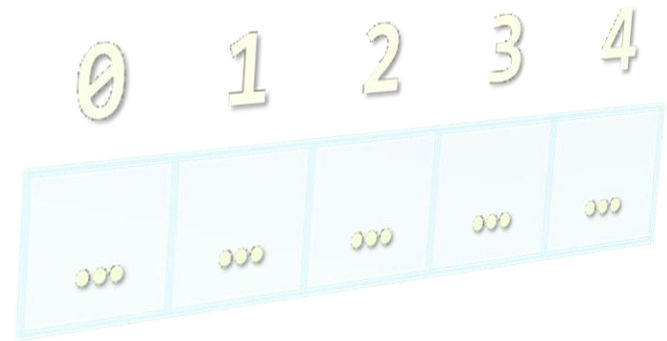
# Kotlin

$\lambda$

# Table of Contents

1. Collections
2. Lambda
3. Common operations on collections
4. Json

# Collections



# Arrays

- Kotlin has a special class called **Array<T>** to declare arrays

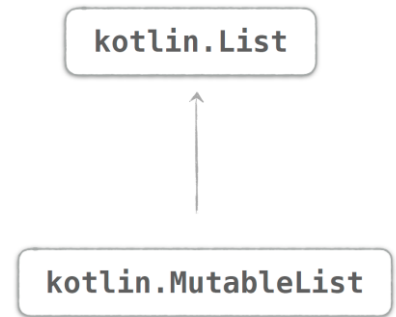
```
val colors: Array<String> = arrayOf("Red", "Green", "Blue")
val names = arrayOf("Ali", "Ahmed", "Sara")
val nulls: Array<String?> = arrayOfNulls(10)
```

```
val nums = arrayOf(2, 3, 4)
val nullNums = arrayOfNulls(10)
```

```
colors.forEach { println(it) }
```

- Better to use **List, Set, Map**

# List



*// Immutable list - cannot add/remove elements*

```
val numsList = listOf(1, 2, 3)
```

*// mutable list - can add/remove elements*

```
val mutableList = mutableListOf(1, 2, 3)
```

```
mutableList.add(4)
```

```
mutableList.removeAt(0)
```

# Set

- *Set is same as List but does not allow duplicates*

*// immutable set and mutable set*

```
val colors = setOf("red", "blue", "yellow")
```

```
val mutableColors = mutableSetOf("red", "blue", "yellow")
```

```
mutableColors.add("pink")
```

```
mutableColors.add("blue") // will not be added
```

# Map

- Stores keys and associated values

```
val languages = mapOf(  
    1 to "Python",  
    2 to "Kotlin",  
    3 to "Java"  
)  
  
for ((key, value) in languages) {  
    println("$key => $value")  
}
```

# arrayOf vs. listOf


- arrayOf creates an array having a sequential fixed-size memory region for storing items of the same type.
- The values of elements of arrayOf can change, the elements of listOf cannot be changed

```
val nums = listOf(1, 3, 5, 6)
// This will not compile
nums[0] = 2
val nums = arrayOf(1, 3, 5, 6)
// This is ok
nums[0] = 2
val nums = mutableListOf(1, 3, 5, 6)
// This is ok
nums[0] = 2
```

- Array have fixed size and cannot expand or shrink
- mutableListOf has add and remove functions to expand/shrink the list
- Good practice is to prefer using lists over arrays, the reasoning is the same to [that for Java](#).



# Lambda

A large, stylized black lambda symbol ( $\lambda$ ) centered on the page.

# Imperative vs. Declarative

## Imperative Programming

- You tell the computer **how** to perform a task

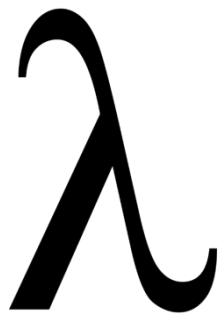
## Declarative Programming

- You tell the computer **what you want**, and you let the compiler (or runtime) figure out the best way to do it. This makes the code simpler and more concise
- Also known as **Functional Programming**
- **Declarative programming using Lambdas helps us to achieve KISS**

**K**EEP **I**T **S**HORT & **S**IMPLE



# What is a Lambda?



- Lambda is very similar to *a function*. It has:
  - Parameters
  - A body
  - A return type
- It **don't have a name** (anonymous method)
- It can be assigned to a variable
- It **can be passed as a parameter** to other function:
  - As *code* to be executed by the receiving function
- Concise syntax:

**{ Parameters -> Body }**

# Passing Lambda as a Parameter

- Lambda expression can be passed as a parameter to methods such as *forEach*, *filter* and *map* methods :

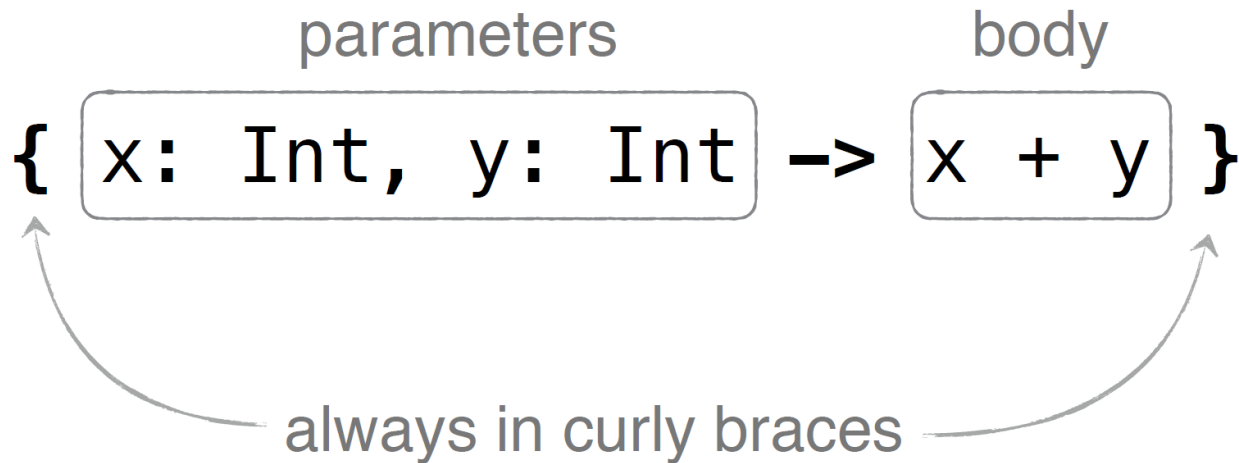
```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)
numbers.forEach { e -> println(e) }
```

**forEach** - Calls a Lambda on Each Element of the list

- Left side of **->** operator is a parameter variable
- Right side is the code to operate on the parameter and compute a result
- When using a lambda with a List the compiler can determine the parameter type

# Lambda

- **Lambda** is an anonymous function that you can store in a variable, pass them as parameter, or return from other function



`list.any({ i: Int -> i > 0 })`

↑  
full syntax

```
list.any() { i: Int -> i > 0 }
```

when lambda is the last argument,  
it can be moved out of parentheses

```
list.any { i: Int -> i > 0 }
```

empty parentheses can be omitted

```
list.any { i -> i > 0 }
```

type can be omitted if it's clear from the context

```
list.any { it > 0 }
```

**it** denotes an argument (if it's only one)

## Lambda Short Form

# Multi-line lambda

```
list.any {  
    println("processing $it")  
    it > 0  
}
```

Last expression is the result



# Lambda usage

- Allows working with collections in a **functional style**

```
val nums = 1..10
//Version 1
var hasEvenNumber = nums.any(isEven)
//Version 2
hasEvenNumber = nums.any { n -> n % 2 == 0 }
//Version 3 - best
hasEvenNumber = nums.any { it % 2 == 0 }

//Version 1
var evens = nums.filter(isEven)
//Version 2
evens = nums.filter { n -> n % 2 == 0 }
//Version 3 - best
evens = nums.filter { it % 2 == 0 }
```



# Lambda usage

e.g. What's the average age of employees working in Doha?



```
val employees = listOf<Employee>(
    Employee("Sara Faleh", "Doha", 30),
    Employee("Mariam Saleh", "Istanbul", 22),
    Employee("Ali Maleh", "Doha", 24)
)
```

```
val avgAge = employees.filter { it.city == "Doha" }
                        .map { it.age }
                        .average()
```

# Member references

```
class Person(val name: String, val age: Int)
```

```
people.maxBy { it.age }
```

 Convert lambda to reference 

```
people.maxBy(Person::age)
```

Class

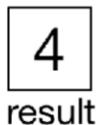
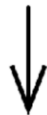
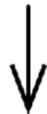
Member

# Collections vs Sequences

## => Eager vs Lazy evaluation

- Eager: each operation produces an intermediate collection having all the results then passes it to the next operation in the pipeline
- Lazy: **no** intermediate collections are created on chained calls

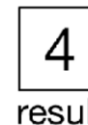
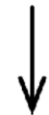
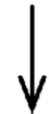
Eager



map

find

Lazy



# Sequence

*// Sequences represent lazily-evaluated collections*

```
val numSequence = generateSequence(1, { it + 1 })
```

*// Nothing happens until terminal operation .toList() is called*

```
val nums = numSequence.take(10).toList()
```

```
println(nums) // => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

*// Convert list to a sequence to enable lazy evaluation*

```
val numbers = listOf(1, 2, 3, 4, 5)
```

```
val sum = numbers.asSequence()
```

```
    .map { it * 2 } // Lazy
```

```
    .filter { it % 2 == 0 } // Lazy
```

```
    .reduce(Int::plus) // Terminal (eager)
```

```
println(sum) // 30
```

# Lazy Evaluation

- Nothing happens until terminal operation is called
- No intermediate collections are created on chained calls

intermediate operations

`sequence.map { ... }.filter { ... }.toList()`

terminal operation

The diagram shows the code `sequence.map { ... }.filter { ... }.toList()`. Above the `map` and `filter` methods, a bracket spans both, with the text "intermediate operations" above it. Two diagonal lines point from this bracket down to the `map` and `filter` methods. Below the `toList()` method, a bracket spans the `toList()` call, with the text "terminal operation" below it. A diagonal line points from this bracket down to the `toList()` method.

# Eager vs. Lazy Evaluation

```
val nums = listOf(1, 2, 3, 4)
nums.map {
    println( it * 2)
    it * 2
}
.find { it == 4 }
```

Output:

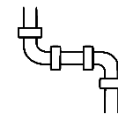
```
val sequence = sequenceOf(1, 2, 3, 4)
sequence.map {
    println( it * 2)
    it * 2
}
.find { it == 4 }
```

Output:

Explanation of the observed difference between outputs:

# Common operations on collections

Filter, Map, Reduce, and others





# Common operations on collections

**.map** 

- Applies a function to each list element

**.filter(condition)** 

- Returns a new list with the elements that satisfy the condition

**.find(condition)** 

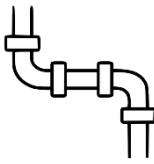
- Returns the first list element that satisfy the condition

**.reduce** 

- Applies an accumulator function to each element of the list to reduce them to a single value



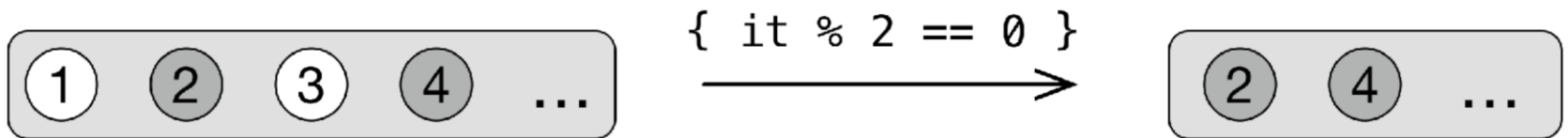
# Operations Pipeline



- **A pipeline of operations:** a sequence of operations where the output of each operation becomes the input into the next
  - e.g., `.filter -> .map -> .sum`
- Operations are either **Intermediate** or **Terminal**
- **Intermediate operations** produce a new list as output (e.g., `map`, `filter`, ...)
- **Terminal operations** are the final operation in the pipeline (e.g., `find`, `reduce`, `sum` ...)
  - Once a terminal operation is invoked then no further operations can be performed

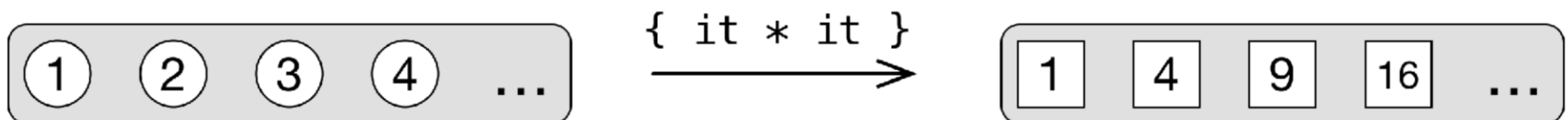
# Filter

Keep elements that satisfy a condition



# Map

Transform elements by applying a Lambda to each element



# Reduce

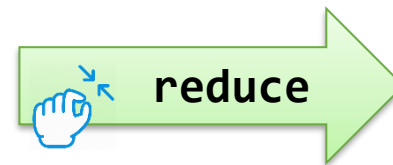
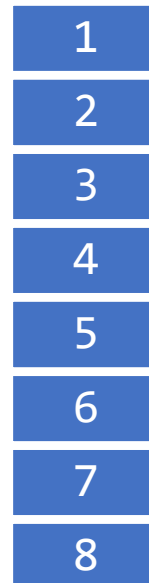


Apply an accumulator function to each element of the list to reduce them to a single value

```
// Imperative
var sum = 0
for(n in numbers)
    sum += n
```

```
//Declarative
var total = numbers.reduce { sum, n -> sum + n }
//Another way with the ability to set the initial
value of sum
total = numbers.fold(0) { sum, n -> sum + n }
//Short form
total = numbers.sum()
```

Collapse the multiple elements of a list into a single element

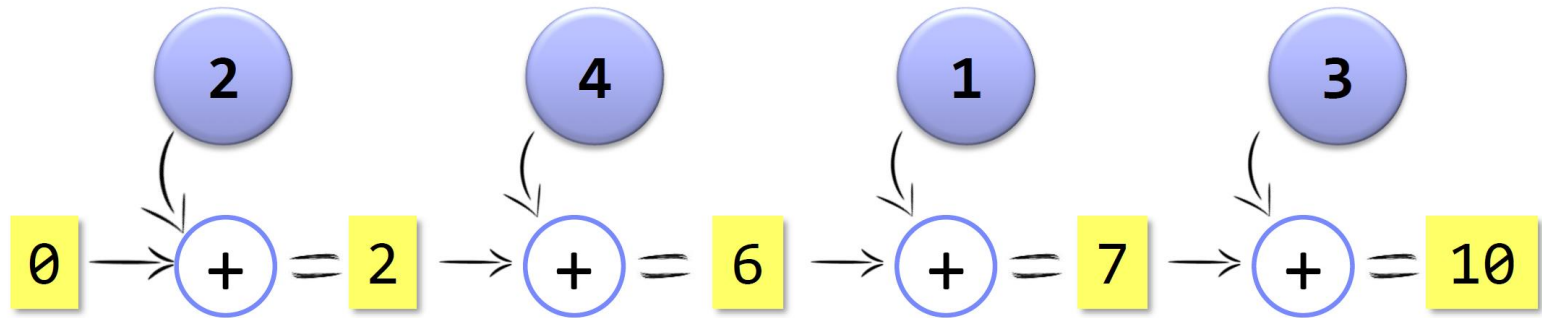


36

Accumulation  
Variable

Accumulation  
Lambda

# Reduce



*.reduce* { sum, n -> sum + n }

Reduce is **terminal** operation that yields a single value

# Convenience Reducers

sum, average, count, min, max

- They are **terminal** operations that yield a single value

```
val nums = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
val sum = nums.sum()
```

```
val count = nums.count()
```

```
val average = nums.average()
```

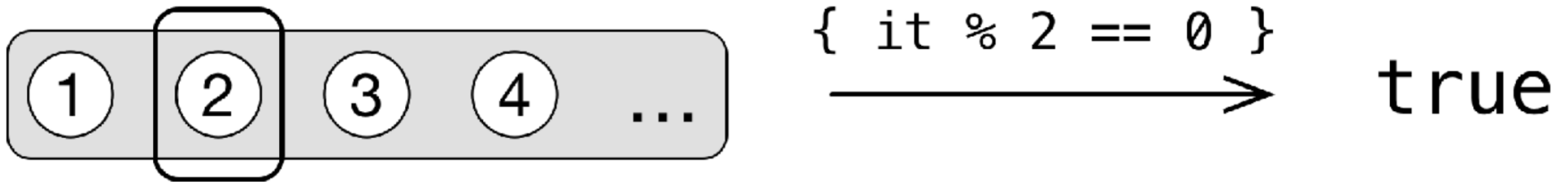
```
val max = nums.maxOrNull()
```

```
val min = nums.minOrNull()
```

# any (all, none)

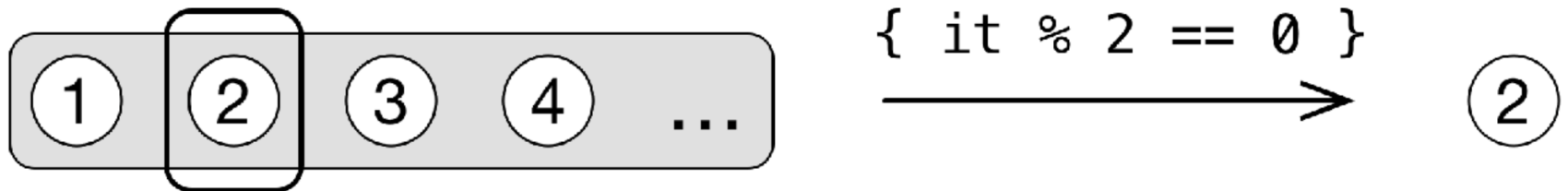


- **any** returns true if it finds an element that satisfies the lambda condition
- **all** returns false if it finds an element that fails the lambda condition
- **none** returns false if it finds an element that satisfies the lambda condition

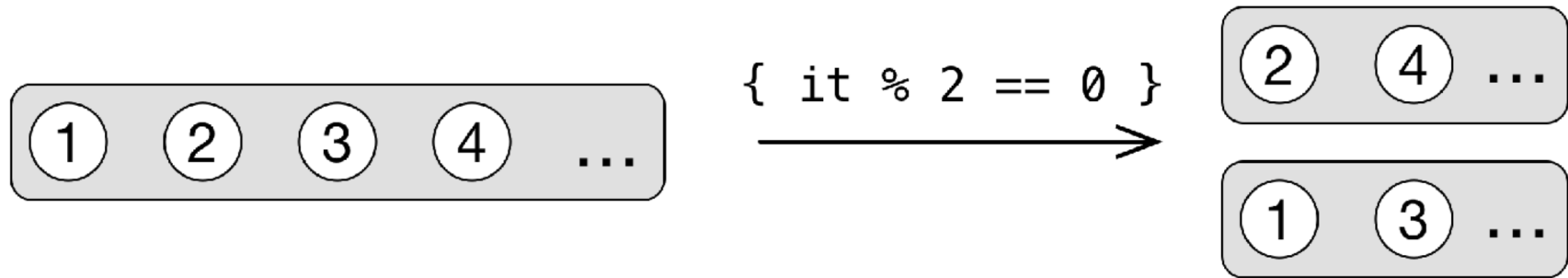


## find / firstOrNull

Return first element satisfying a condition



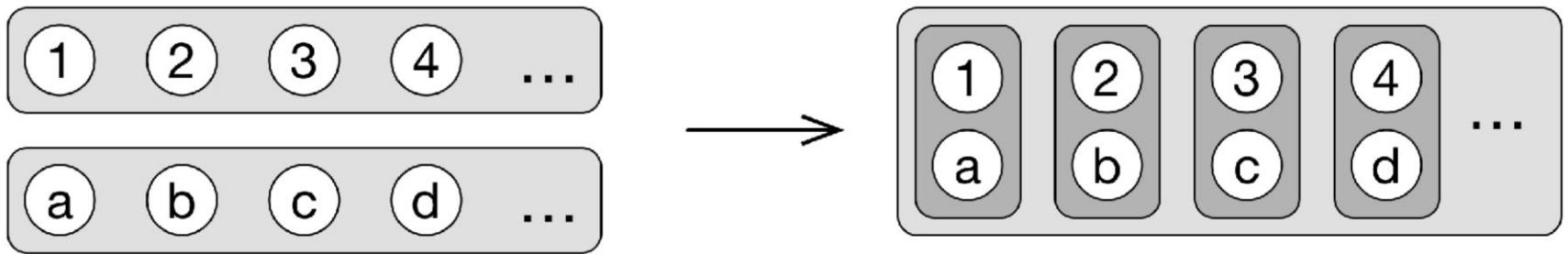
# partition



```
val (even, odd) = listOf(1, 2, 3, 4).partition { it % 2 == 0 }
```

# zip

Returns a list from the elements from 2 lists having the same index.  
The resulting list ends as soon as the shortest input list ends



```
val nums = listOf(1, 2, 3, 4)
```

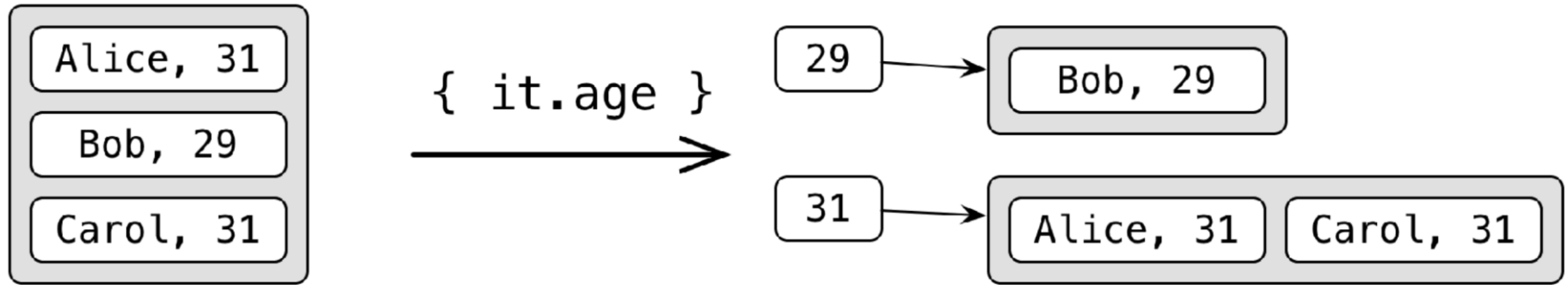
```
val letters = listOf("a", "b", "c", "d")
```

```
val result = nums.zip(letters)
```



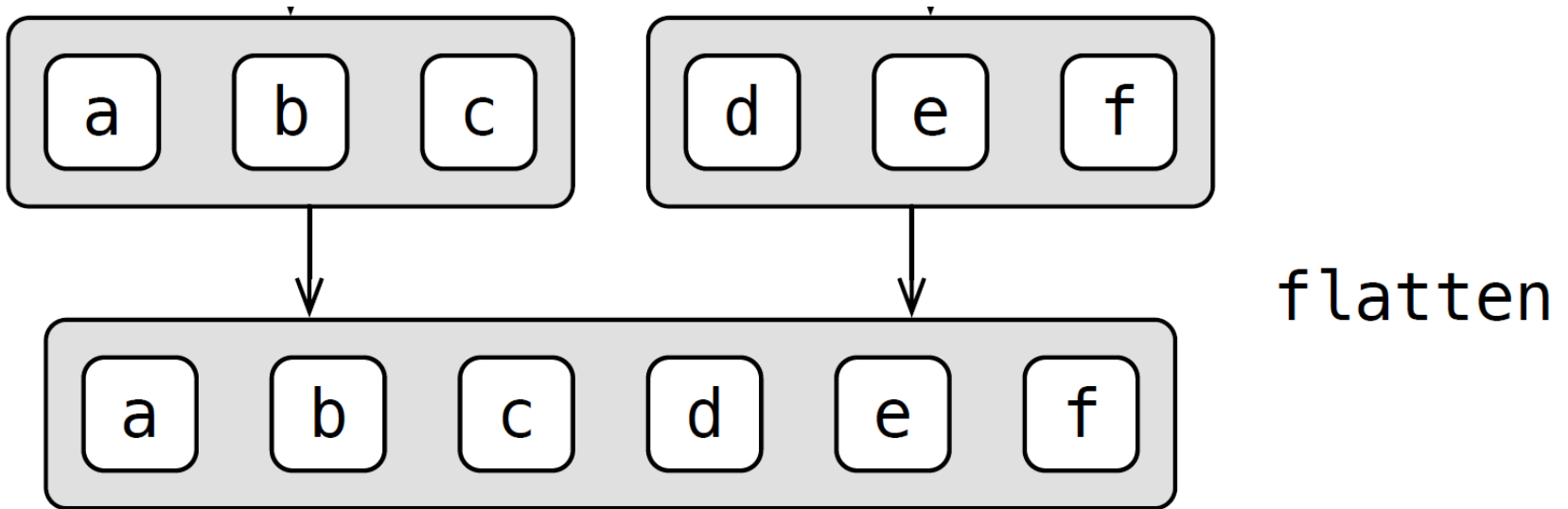
# groupBy

- groupBy is used to split a list into groups



```
people.groupBy { it.age }
```

# flatMap



```
val listOfList = listOf(  
    listOf("a", "b", "c"),  
    listOf("d", "e", "f")  
)  
val singleList = listOfList.flatMap { it }
```

# flatMap

Do a map and flatten the results into 1 list

Each book has a list of authors. **flatMap** combines them to produce a single list of **all** authors

```
class Book(  
    val title: String,  
    val authors: List<String>  
)  
  
fun main() {  
    val books = listOf(  
        Book("Head First Kotlin", listOf("Dawn Griffiths", "David Griffiths")),  
        Book("Kotlin in Action", listOf("Dmitry Jemerov", "Svetlana Isakova"))  
    )  
    val authors = books.flatMap { it.authors }  
}
```

# Sort a List using Lambda

Sort strings by length (shortest to longest)

```
val names = listOf("Abderahame", "Abdelkarim", "Ali", "Sarah", "Samira", "Farida")
println(">Sorted by length:")

var sorted = names.sortedBy { it.length }
println(sorted)

println("\n>Sorted by length and then alphabetically:")
//Sort strings by length (shortest, longest) and then alphabetically
sorted = names.sortedWith( compareBy( { it.length }, { it }) )
println(sorted)
```

# Use **.compareBy** for multi-step comparisons

```
class Person(val name: String,  
             val age: Int)
```

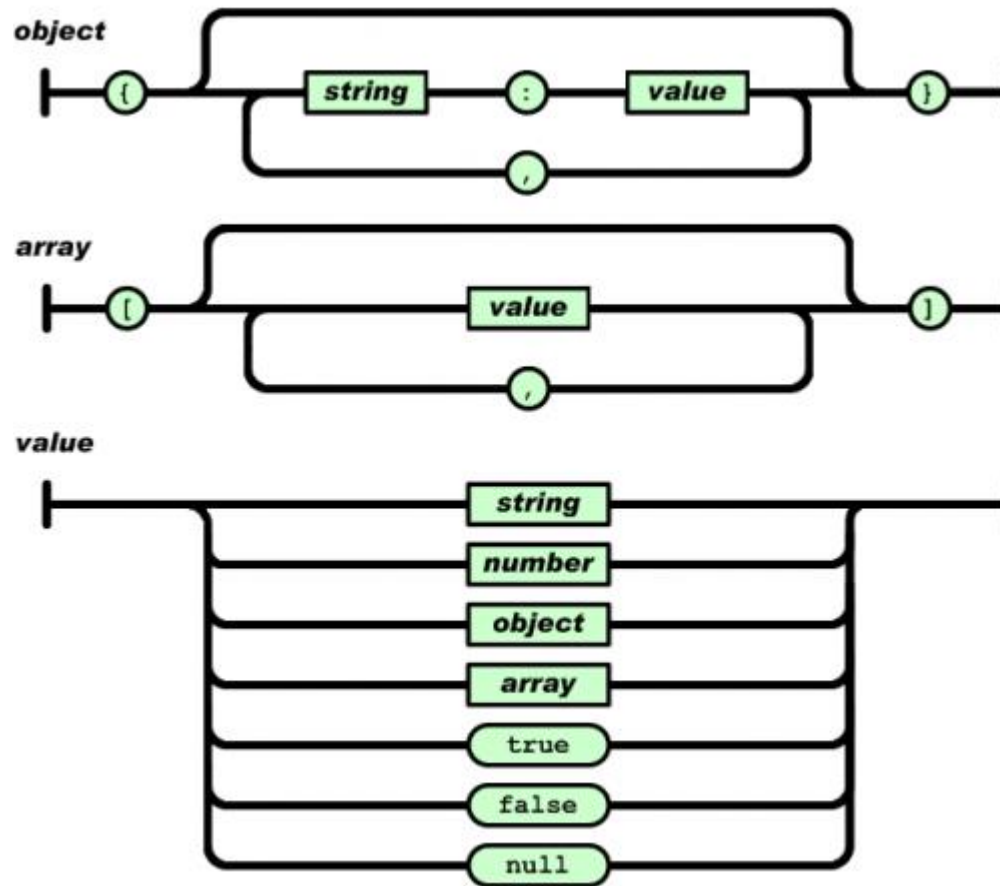
```
fun sortPersons(persons: List<Person>) =  
    persons.sortedWith(  
        compareBy(Person::name,  
                  Person::age))
```

# Use **.apply** for object initialization or for changing multiple attributes

```
val conference = Conference( name: "Kotlin Conf.", city: "Istanbul", fee: 300.0)
```

```
// Version 1 🗨️ - Change the conference city and fee then print it  
conference.city = "Doha"  
conference.fee = 200.0  
println(conference)
```

```
// Version 2 ** Best 👍 ** - Change the conference city and fee then print it  
// .apply changes the object and returns it  
// .also execute some processing on the object and returns it  
conference.apply { this: Conference  
    | city = "Doha"  
    | fee = 200.0  
}.also { println(it) }
```



# JSON Data Format

- **JSON** (JavaScript Object Notation) is a very popular **lightweight data format** to transform an object to a **text** form to ease storing and transporting data
- **Json** class could be used to transform an object to json or transform a json string to an object

Transform an instance of Surah class to a JSON string:

```
val fatiha = Surah(1, "الفاتحة", "Al-Fatiha", 7, "Meccan")
val surahJson = Json.encodeToString(fatiha)
```

*// Converting a json string to an object*

```
val surah = Json.decodeFromString<Surah>(surahJson)
```



```
{
  "id": 1,
  "name": "الفاتحة",
  "englishName": "Al-Fatiha",
  "ayaCount": 7,
  "type": "Meccan"
}
```

Surah
id: int
name: String
englishName: String
ayaCount: int
type: String



# @Serializable

- To use Json sterilization the class must be annotated with **@Serializable**

**@Serializable**

```
data class Surah (  
    val id : Int,  
    val name: String,  
    val englishName : String,  
    val ayaCount : Int,  
    val type: String  
)
```

# Read JSON file

- Read a JSON file and convert its content to objects

```
val filePath = "data/surahs.json"
```

```
val fileContent = File(filePath).readText()
```

```
val surahs = Json.decodeFromString<List<Surah>>(fileContent)
```



You may use <https://plugins.jetbrains.com/plugin/10054-generate-kotlin-data-classes-from-json> Android Studio plugin to generate a Kotlin class from a json string!

# Dependencies to use Kotlin Serialization

- To be able use **@Serializable** and **Json** class you need to add these dependencies then sync:

1) Add this apply plugin to the build.gradle of the module

```
id 'org.jetbrains.kotlin.plugin.serialization' version '1.5.20'
```

3) Add to dependencies to the build.gradle of the module

```
implementation 'org.jetbrains.kotlinx:kotlinx-serialization-json:1.2.2'
```

# Summary

- To start thinking in the functional style ***avoid loops*** and instead use Lambdas
  - Widely used for list processing and GUI building to handle events
- A list can be processed in a pipeline
  - Typical pipeline operations are filter, map and reduce
- JSON is a very popular lightweight data format to **transform an object to a text form** to ease storing and transporting data